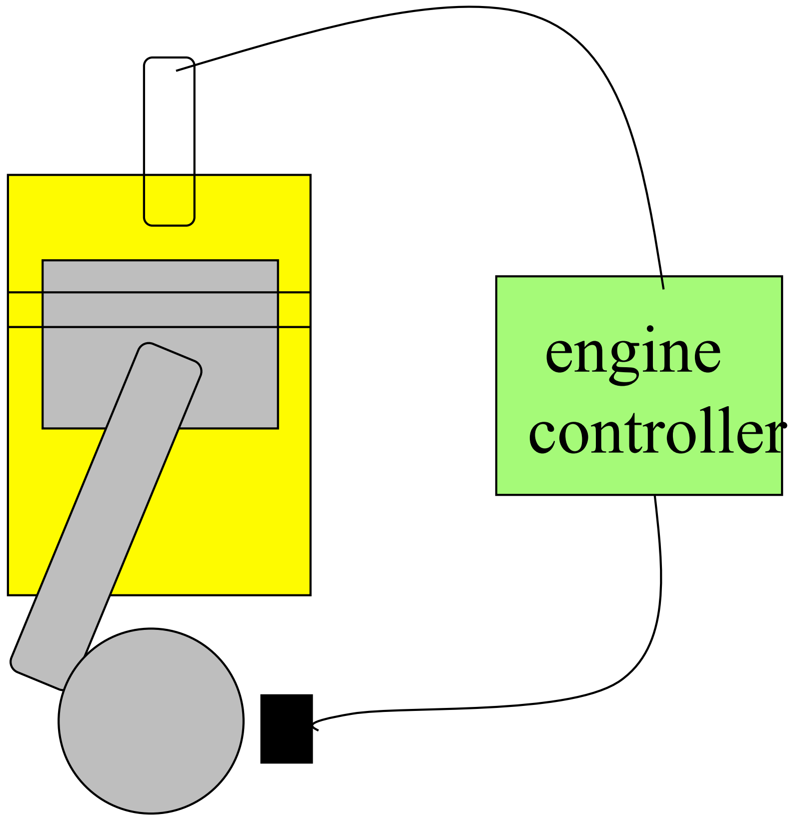
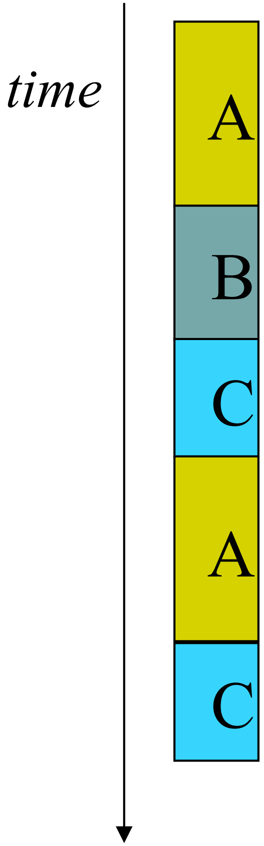
Example: engine control

- Tasks:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter
  - state machine



Life without processes

- Code turns into a mess:
  - interruptions of one task for another
  - spaghetti code




```
A_code();
…
B_code();
…
if (C) C_code();
…
A_code();
…
switch (x) {
 case C: C();
 case D: D();
..
```

Managing multitasking

- Co-routines.
- Co-operative multitasking.
- Preemptive context switching.
- Interrupts.

Co - routine methodology

- Like subroutine, but caller determines the return address.
- Co-routines voluntarily give up control to other co-routines.
- Pattern of control transfers is embedded in the code.

Co - routines

ARM assembly code shown: r15 is the program counter, r13 stores co-routine 1 return address and r14 for co-routine 2.

Initialise:

```
ADR r14,co2a
```

Co-routine 1

```
co1a …
    ADR r13,co1b
    MOV r15,r14
co1b …
    ADR r13,co1c
    MOV r15,r14
co1c ...
```

Co-routine 2

```
co2a …
    ADR r14,co2b
    MOV r15,r13
co2b …
    ADR r14,co2c
    MOV r15,r13
co2c …
```

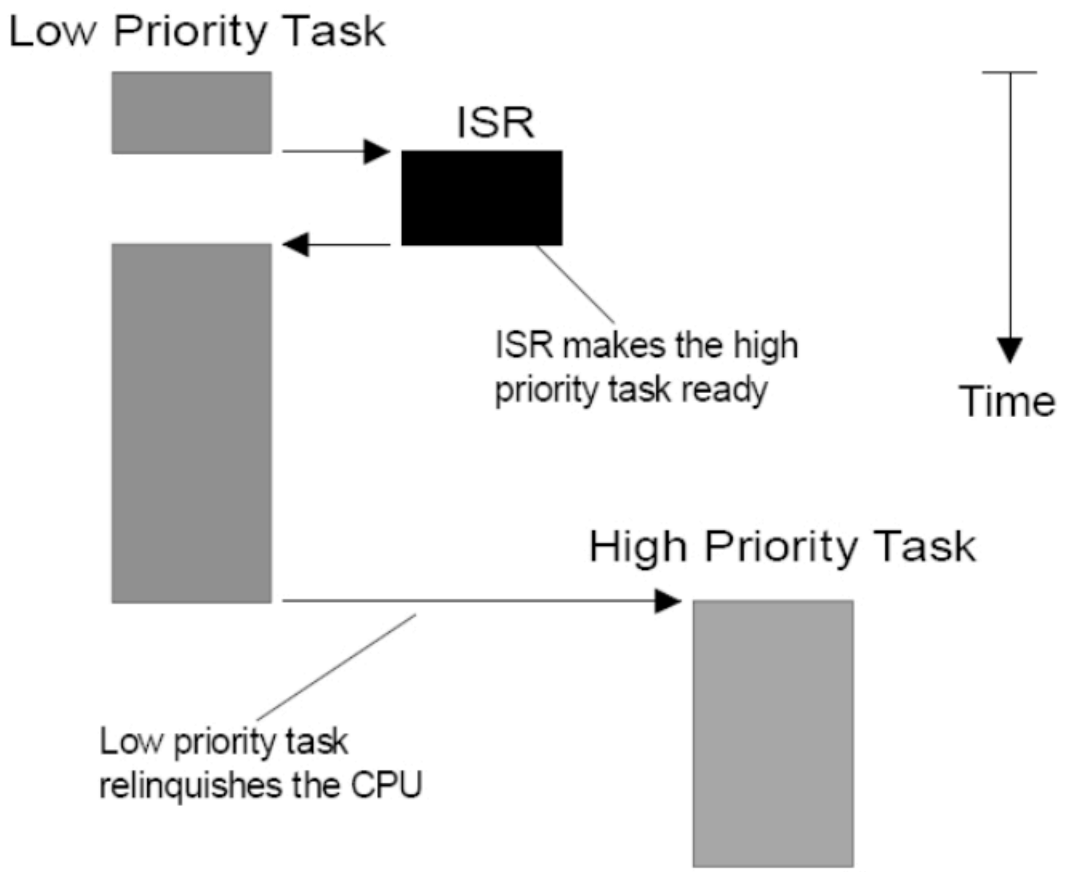
Co - operative multitasking

- Each process allows a context switch at cswitch() call.
  - still relies on processes to give up CPU.
- Separate scheduler chooses which process runs next.

Co - operative multitasking

```
void task1() {
    while (1) {
        /* do some work */
        cswitch()
        /* do some work */
        cswitch()
    }
}
```

```
void task2() {
    while (1) {
        /* do some work */
        cswitch()
        /* do some work */
        cswitch()
    }
}
```



From www.micrium.com

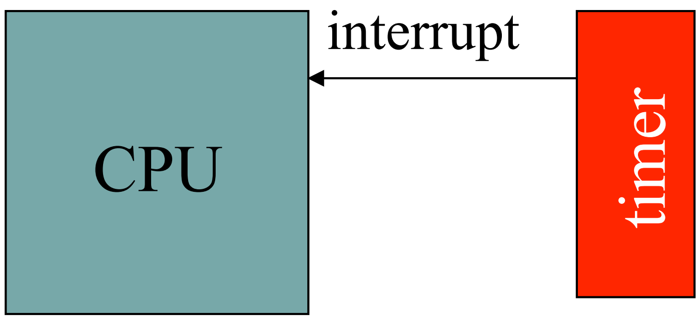
Figure 2. Non-preemptive scheduling

Problems with co - operative multitasking

- When a high priority process "wakes" it has to wait for a lower process to call OS.
  - Results in longer response times to events
  - => higher latency.
- Programming errors can keep other processes out:
  - process never gives up CPU;
  - process waits too long to switch, missing input.

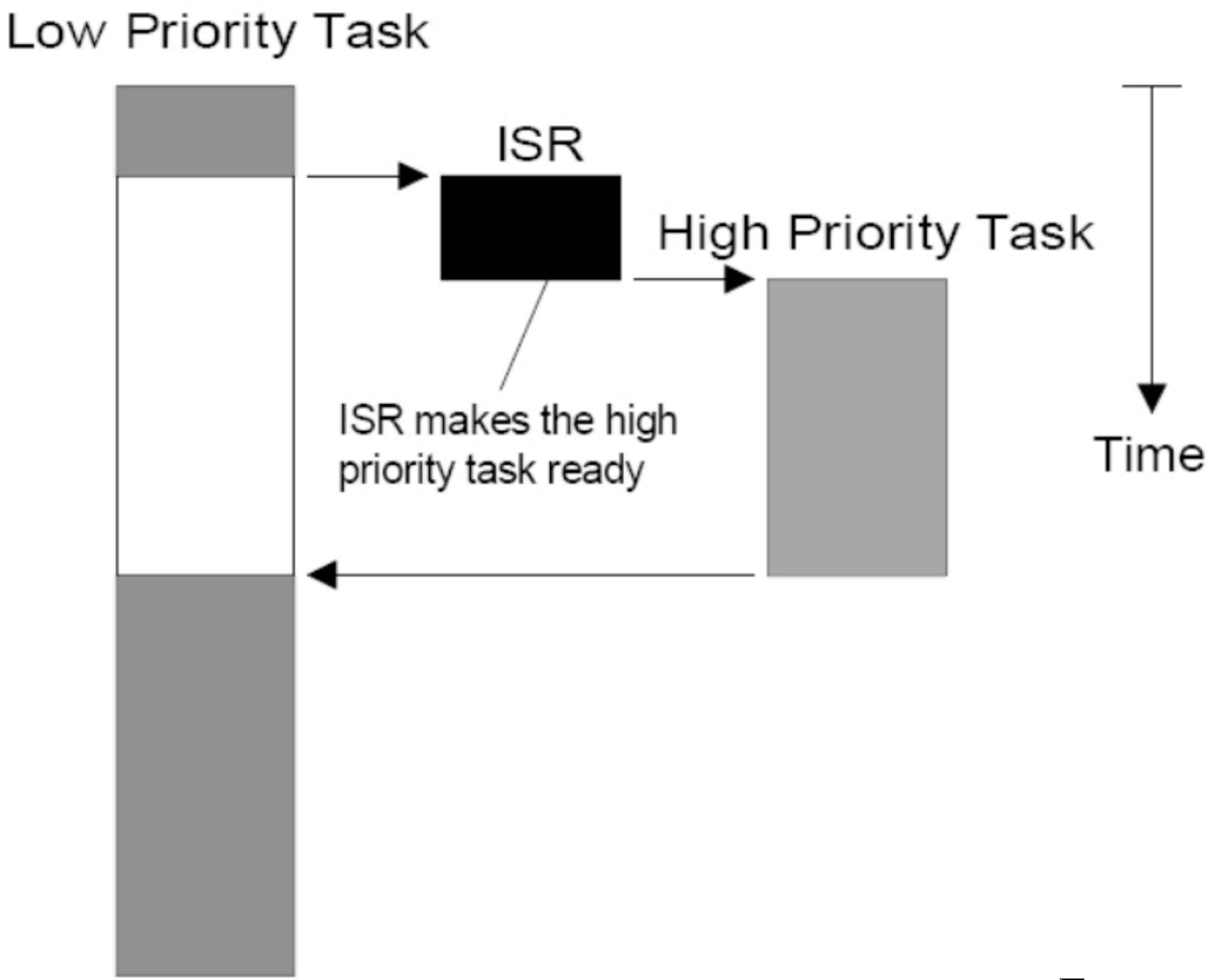
Preemptive multitasking

- OS controls when contexts switches:
- Use timer/other interrupts to call OS.
- OS determines what process runs next.



Preemptive context switching

- Timer interrupt gives control to OS, which saves interrupted process's state in an activation record.
- OS chooses next process to run – called scheduling .
- OS installs desired activation record as current CPU state.
- uC/OS -II is a preemptive real time kernel, with deterministic execution times, portable code (eg runs on NIOS -II)



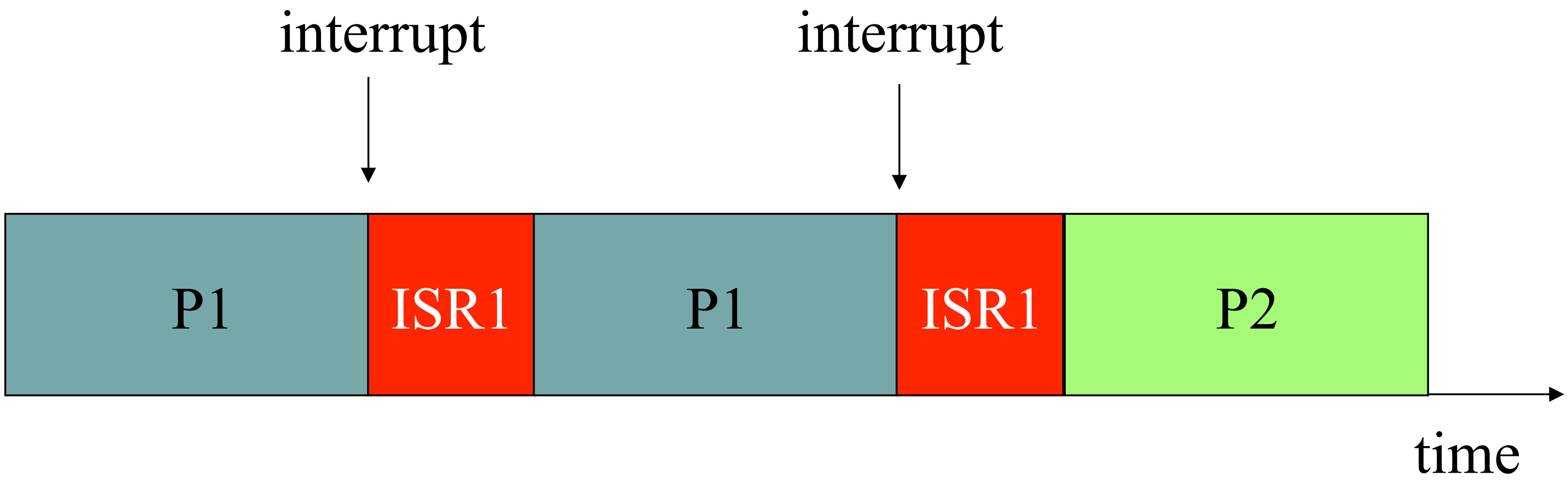
From www.micrium.com

Figure 3. Preemptive scheduling

Interrupts

- Multiple timer interrupts, with different rates:
- Every ISR performs context switch

Flow of control with interrupts

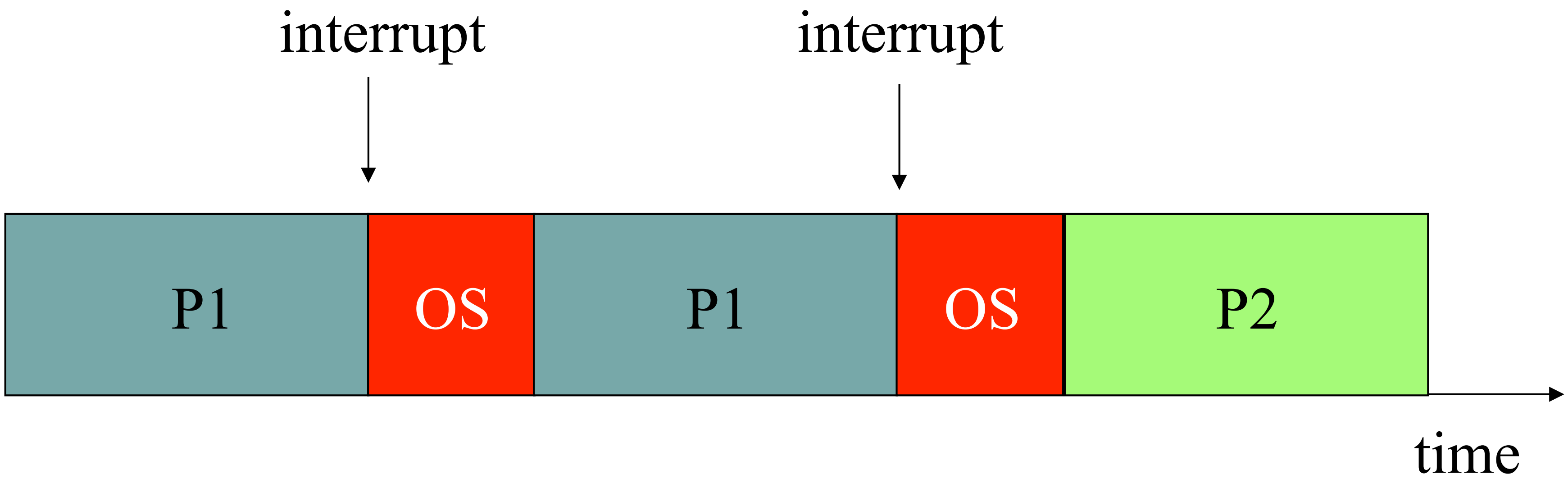


Interrupts

```
void task1() {
    while (1) {
        /* do some work */
    }
}
```

```
void task2() {
    while (1) {
        /* do some work */
    }
}
```

Flow of control with preemption

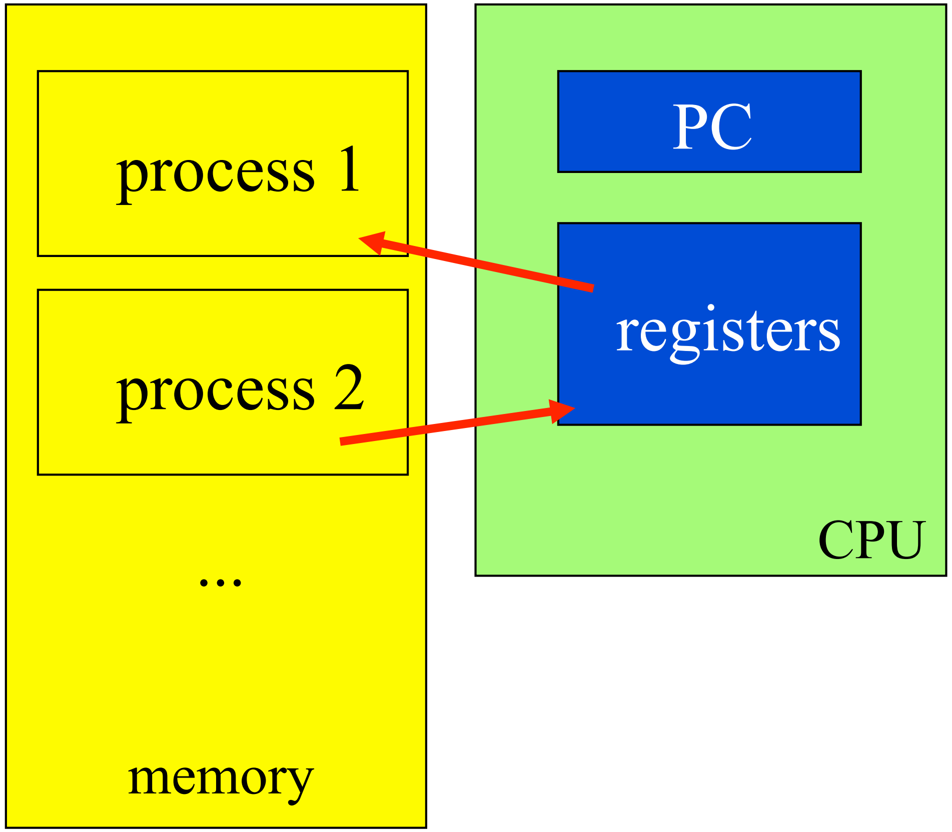


Managing multitasking

- Co-routines.
- Co-operative multitasking.
- Preemptive context switching.
- Interrupts.

Processes and CPUs

- Activation record: copy of process state.
- Context switch:
  - current CPU context goes out;
  - new CPU context goes in.



Context switching

- Must copy all registers to activation record, keeping proper return value for PC.
- Must copy new activation record into CPU state.
- How does the program that copies the context keep its own context?

Context Switch

- Save the registers of the current task onto its stack.(return address, frame pointer and currently used set….r16-r23)
- Save the current task stack pointer into its Task Control Block(TCB)
- Find the highest priority ready task and find its TCB
- Get the new task stack pointer from its TCB
- Restore the registers of the new task from the stack
- Start executing the new task

Context Switch in NIOS II using μ C/OS- II

```
/* Save the remaining registers to the stack. */
    addi sp, sp, -44

    ldw r4, %gprel (OSTCBCur)(gp)
            /* %gprel(immed32) = immed32- gp, gp=global pointer register of NIOS */

    stw ra,  0(sp)
    stw fp,  4(sp)
    stw r23, 8(sp)
    stw r22, 12(sp)
    stw r21, 16(sp)
    stw r20, 20(sp)
    stw r19, 24(sp)
    stw r18, 28(sp)
    stw r17, 32(sp)
    stw r16, 36(sp)

    /* Save the current tasks stack pointer into the current tasks OS_TCB.
    * i.e. OSTCBCur ->OSTCBStkPtr = sp;
    */
                   /* save the stack pointer (OSTCBStkPtr */
    stw sp, (r4)   /* is the first element in the OS_Task Control Block  */
                   /* structure.                        */
```

```
/* OSTCBCur = OSTCBHighRdy; */
/* OSPrioCur = OSPrioHighRdy;  */

   ldw r4, %gprel (OSTCBHighRdy)(gp)
   ldb r5, %gprel (OSPrioHighRdy)(gp)

   stw r4, %gprel (OSTCBCur)(gp)       /* set the current task to be the new task */
   stb r5, %gprel (OSPrioCur)(gp)      /* store the new task's priority as the current */
                                       /* task's priority                           */

   /* Set the stack pointer to point to the new task's stack */
   ldw sp, (r4)          /* the stack pointer is the first entry in the OS_TCB structure */

   /* Restore the saved registers for the new task. */
   ldw ra,  0(sp)
   ldw fp,  4(sp)
   ldw r23, 8(sp)
   ldw r22, 12(sp)
   ldw r21, 16(sp)
   ldw r20, 20(sp)
   ldw r19, 24(sp)
   ldw r18, 28(sp)
   ldw r17, 32(sp)
   ldw r16, 36(sp)

   addi sp, sp, 44

   /* resume execution of the new task. */
   ret
```

Terms

- Thread = lightweight process: a process that shares memory space with other processes.
  - Linux : uses POSIX (Portable Operating System Interface) threads called pthreads.
  - uC/OS -II (Micro -controller Operating System): threads are called tasks and the operating system is called a real time kernel
- Reentrancy:

  ability of a program to be executed concurrently in different processes with the same results.

Tasks in uC/OS- II

- Tasks must have ***unique*** priority
  - Highest priority ready task always runs.
  - lower number represents higher priority – eg 0 highest priority.
  - Idle task is given lowest priority eg 63 in a 64 task system.
- Each task has its own stack:
  - stores return addresses for functions/ISRs,
  - local variables

Create a task with:

error = OSTaskCreate(task, pdata, pstack, priority);

Function name that starts task execution

Pointer to stack

Unique priority

Always check for errors

Parameter to function

See uC/OS-II reference guide on unit webpage